Combinable Memory-Block Transactions

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Phillip B. Gibbons
Intel Research Pittsburgh
Pittsburgh, PA, USA
phillip.b.gibbons@intel.com

S. Harsha Vardhan
Carnegie Mellon University
Pittsburgh, PA, USA
harshas@cs.cmu.edu

ABSTRACT

This paper formalizes and studies *combinable memory-block transactions (MBTs)*. The idea is to encode short programs that operate on a single cache/memory block and then to specify such a program with a memory request. The code is then executed at the cache or memory controller, atomically with respect to other accesses to that block by this or other processors. The combinable form allows combining within the memory system or network. In addition to allowing for the standard set of read-modify-write operations (e.g., test-and-set, compare-and-swap, fetch-and-add), MBTs can be used to define other useful operations—such as a fetch-and-add that does not decrement below zero.

We show how MBTs can be used to design simple and efficient implementations of a variety of protocols and algorithms, including a priority write, a semaphore with a non-blocking P operation, a bounded queue, and a timestamp-based transactional memory system. In all cases the protocols gain some advantage by using MBTs that are different from the standard set of operations. To gain an understanding of the efficiency that can be gained by using combining, we define a notion of bounded contention and show that all our protocols have bounded contention under arbitrary loads.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; B.3.2 [**Memory Structures**]: Design Styles—*Shared memory*

General Terms

Algorithms, Design, Performance, Theory

Keywords

memory-block transactions, combining, shared memory, linearizability, read-modify-write, priority write, semaphore, queue, stack, transactional memory, contention

SPAA'08, June 14–16, 2008, Munich, Germany.

1. INTRODUCTION

There has been a long history of research on and implementation of instructions that operate atomically on a single word or block of memory, including the well-studied swap, compare-and-swap (CAS), test-and-set (TS), and fetch-and-add (FA) operations. It is well understood that these operations can greatly simplify implementing efficient versions of certain protocols and problems [8, 16, 21, 9]. Hardware support for some of these operations dates back at least to the 1970s with the IBM 370 [11], which supported compare-and-swap. Work on the Ultracomputer further showed how combining can be used to efficiently support some of these operations even when many processors concurrently access the same location [8, 7].

Unfortunately, there does not appear to be any small universal set of operations that efficiently simulate all atomic block operations. Known universality results [9] are neither efficient in practice nor in theory [13, 5]. Often a user needs some variant or extension of an operation and there is no satisfactory way to simulate it given the existing set. For example, one might want an atomic non-negative fetch-and-decrement (NN-FD), which differs from the standard fetch-and-decrement by never decrementing the counter below zero (we give an example of such a use in Section 4.2). It is not known how to simulate an NN-FD with a standard FA in an efficient, fair, linearizable and lock-free manner, even given other operations such as a CAS. Various meta-operations can be used to implement atomic operations, such as the load-linked store-conditional (LL-SC) instructions or transactional memory. These, however, have their own problems especially under high contention. Most implementations do not support fair access, many implementations do not guarantee any progress, and none permit combining.

Recently Fang et al. [4] have proposed what they refer to as active memory operations (AMOs). The idea is to encode small programs that operate on memory and to specify a program with a memory request. The code is then executed at the memory controller. Based on detailed simulation they show that compared to using LL-SC, AMOs can greatly speed up certain tasks. The authors consider both atomic transactions on a single block and streaming operations across a strided vector within a single memory controller. Here we are just interested in the memory-block transactions (MBTs). In addition to allowing for the standard set of operations (*e.g.*, CAS, TS, FA), MBTs can be used to define others—such as an NN-FD. MBTs certainly do not solve all concurrency problems—in particular those involving multiple dispersed blocks of memory—but they do,

in principal, allow for an arbitrary atomic update on a single memory block. MBTs, however, do not combine and therefore still sequentialize access to highly-contended memory locations.

In this paper we formalize MBTs and extend them to support combining using the notion of *combinable memory-block transactions*. As with the combining suggested for the Ultracomputer [8, 16], the idea is that requests for the same location can be combined in the network before reaching the memory itself. In fact they could be combined at various levels in the memory hierarchy. For example in a two-level hierarchical cache coherence scheme [20], combining might be applied between the two levels to combine requests from the same lower level node to the next level. A combinable MBT specifies two functions that are executed in the memory system, one that is used for combining and a second that is used for applying the combined request at the memory. Being able to specify the combining code enables a much broader set of operations than the standard set.

We show how MBTs can be used to design simple and efficient implementations of a variety of protocols and algorithms, including a priority write, a semaphore with a non-blocking P operation, a bounded queue, and a timestamp-based transactional memory system (details in Section 4). In all cases the protocols use MBTs that are different from the standard set to gain some advantage. All the MBTs used are combinable. We define a notion of *bounded contention* within our framework, and show that all our protocols have bounded contention under arbitrary loads. We note that many of these protocols are interesting and possibly of utility even in their non-combinable form.

In the remainder of the paper, we begin in Section 2 by formalizing MBTs. Section 3 presents combinable MBTs. Section 4 presents our example protocols and algorithms. Section 5 highlights related work. Finally, Section 6 discusses implementation issues.

2. MEMORY-BLOCK TRANSACTIONS

We assume memory consists of a sequence of words indexed by integer locations. A *memory block* is a constant number of consecutive words in memory. In practice a block should be contained in a single cache line (in this paper the largest block we use has four words). A *memory-block transaction* (MBT) specifies a memory block b, an input $i \in I$, and a transition function $\phi : S \times I \rightarrow S \times O$, where S is the set of states that the memory block can have, and O is a set of possible outputs. The effect of the transaction is to apply ϕ to i and the state of the memory block, update the block with the new state and return the output to the processor requesting the MBT. We assume each transaction x is *invoked* at some time t_x and *responds* with the output at some later time t'_x. We allow for a processor to have multiple outstanding transactions. An invoked transaction (or operation) x is *completed* at time t if $t'_x \leq t$; otherwise, it is *pending*.

We assume that memory-block transactions are linearizable [10] and will also show linearizability for the protocols and algorithms we define. Linearizability effectively means that each operation must act like it happens atomically some time between its invocation and the response. More formally, a history $\mathcal{H}$ is a set of operations (or transactions) each with an invoke time t and, unless it is pending, a response time t'. The operations can come from the same or different processors. Consider the partial order on $\mathcal{H}$ defined as $o_1 <_{\mathcal{H}} o_2$ iff $t'_{o_1} < t_{o_2}$ (o_1 responds before o_2 is invoked). The history $\mathcal{H}$ is *linearizable* if there is some subset $\mathcal{H}'$ of $\mathcal{H}$ and some total order on $\mathcal{H}'$ such that (1) $\mathcal{H}'$ contains all completed operations in $\mathcal{H}$ and possibly some pending ones, and (2) the total order is consistent with both the partial order $<_{\mathcal{H}}$ and the sequential semantics of the operations.[1] When there are no pending operations, $\mathcal{H}' = \mathcal{H}$. We say that an implementation, algorithm, or protocol is linearizable if all the histories it can generate are linearizable. We say that two operations or transactions are *concurrent* if they are not ordered by $<_{\mathcal{H}}$.

```
stype = itype = otype = int

atomic int FA(int *l, int a) {
  int v = *l;
  *l = *l + a;
  return v; }

stype = itype = otype = ptype = int

atomic int CAS_v(int v, int *l, int vn) {
  if (*l == v) {
    *l = vn;
    return 1; }
  else return 0; }
```

Figure 1: MBTs for fetch-and-add (FA) and parameterized compare-and-swap ($\texttt{CAS}_v$).

In this paper we define the transition functions ϕ in C code using an interface of the form:

```
atomic otype phi(stype *, itype)
```

where `stype` is the type for the state (S), `itype` is the type for the input (I), and `otype` is the type of the output (O). (An additional argument type, `ptype`, is defined below.)

For reasons that will become clear when we define combinable memory-block transactions, we will sometimes parameterize (curry) the transition function. We define a class of transition functions as $\{\phi_v : v \in V\}$ where V is the set of possible values for one of the inputs of the operation. For example, consider a compare-and-swap operation $\text{CAS}(l, v, v')$, which takes a pointer to a memory location l and if $v = *l$, swaps v' into l returning 1, and otherwise returns 0. We parameterize this based on v, giving a set of functions $\text{CAS}_v(l, v')$, each of which we refer to as a *single-valued* CAS_v. When parameterizing in sample code we include the parameter as an argument (of type `ptype`) and underline it. Figure 1 gives examples of the definitions of integer FA and CAS using this notation. In both these cases all the types are the same.

3. COMBINABLE MBTS

We now consider conditions under which an MBT is combinable. We partition the transition function ϕ into four functions e, f, g and h, where f and g are sent to the memory system and e and h can be executed at the processor requesting the MBT. We first partition the transition function ϕ into functions $\phi' : S \times I \rightarrow S$, and $h : S \times I \rightarrow O$—*i.e.*,

[1] This is slightly different from the Herlihy and Wing definition [10] in that we allow interleaving of transactions within a processor and therefore drop the assumption of a total order within each processor.

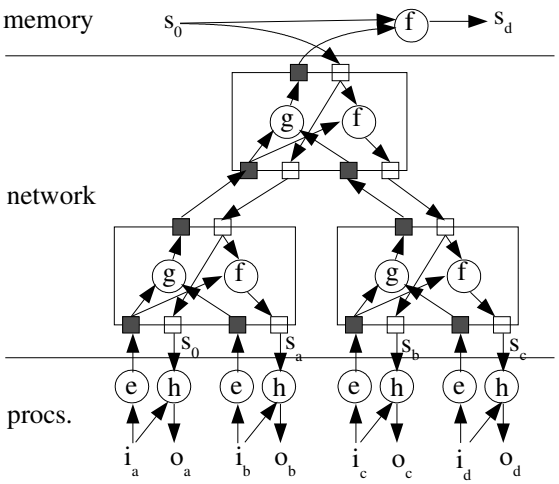

Figure 2: Combining on a tree using e, f, g, and h. This example tree has three internal nodes, represented as rectangles, and four leaves. In general, any subset of the processors might be involved in the combining and the tree need not be balanced.

one that computes the next state and one that computes the output. We say the transition function ϕ is *combinable* if for some set I' there are functions $e : I \to I'$, $f : S \times I' \to S$ and $g : I' \times I' \to I'$ such that:

projection. $\forall s \in S, i \in I : f(s, e(i)) = \phi'(s, i)$,

combining. $\forall s \in S,\ i_1, i_2 \in I' :$
$f(f(s, i_1), i_2) = f(s, g(i_1, i_2))$,

associativity. $\forall i_1, i_2, i_3 \in I' :$
$g(g(i_1, i_2), i_3) = g(i_1, g(i_2, i_3))$, and

bounded size. $|I'| = |S|^k$, for some constant k.

With these definitions a set of transactions using a common ϕ can be combined in a tree as in Figure 2. The reason for adding an additional type I' is that in some cases it is necessary to do some transform on the input to make the transition function combinable. In many cases, as we will see, $I = I'$. The bounded size condition ensures that the values from I' that are passed up the combining tree are at most a constant factor larger than the state. We note that without this condition, or some other condition on space or time, any state transition function is combinable—the function g could simply compose a list of inputs by concatenating its inputs, and the function f could then apply ϕ' across the inputs.

A *combinable memory-block transaction* is a memory-block transaction that uses a combinable transition function. A processor's request of an MBT with a memory block b, an input $i \in I$, and a combinable transition function ϕ is processed using the four functions e, f, g and h associated with ϕ. Namely, the memory system is sent b, $e(i)$, f and g. It processes the request in conjunction with possibly other concurrent requests, using some form of a combining tree, as exemplified in Figure 2. Upon return of a state s from the combining tree, the output of the request is computed as $h(s, i)$.

In many of the examples we will consider, $S = I'$ and $f = g$, so that only one function (f) needs to be sent to the memory system. For example, for the well-known fetch-and-ψ operations for an associative binary operation ψ [8] we have $S = I'$ and $f = g = \psi$. Furthermore, for the fetch-and-ψ, e is the identity function and h is the identity projection on the state (i.e., $\forall s \in S, i \in I : h(s, i) = s$).

Whether transactions combine or not, we refer to the point in time at which the function f is applied to the memory block as the *commit* time, and assume that all commits happen atomically. We say that two block transactions are *compatible* if their memory blocks and combinable transition function ϕ are the same.

THEOREM 1. *Any history H of memory-block transactions in which concurrent compatible transactions can combine using an arbitrary binary tree with e, f, g and h as defined is linearizable.*

PROOF. Because commits for the function f at the memory happen atomically, we can group the transactions by whether they combine and order the groups by the atomic commit. We therefore consider each group of combining transactions.

Consider a combining tree with combining function ϕ, and the associated combining functions e, f, g and h. For every internal node in the combining tree, define the left child to be the child to which it outputs the state it received from its parent (or the memory in case of the root of the tree) without applying function f, and define the right child to be the child to which it sends the state after applying the function f. This definition imposes a natural left to right ordering on the $k \geq 1$ leaves of a tree, where each leaf is an individual transaction. Label the leaves (transactions) in that order from t_1 to t_k.

We will prove that the transactions are linearizable in this ordering, i.e., that the following properties are true:

1. $o_j = h(s_{j-1}, i_j)$, where s_j is defined inductively as $\phi'(s_{j-1}, i_j)$, s_0 being the initial memory state and i_j and o_j the input and the output of transaction t_j, and
2. the state of the memory after all the transactions are committed is s_k.

Claim 1: If the subtree rooted at node n contains leaves l_a through $l_c, c \geq a$, the combined input $i_{a:c}$ forwarded by the node n to its parent is such that $f(s_{a-1}, i_{a:c}) = s_c$.
Proof by induction on the height of the node: The hypothesis is clearly true of all leaves because by projection $f(s_{a-1}, e(i_a)) = \phi'(s_{a-1}, i_a) = s_a$. Suppose the hypothesis is true of both of the combined inputs $i_{a:b}$ and $i_{b+1:c}$ forwarded by the left and right children of a node respectively, where the left tree contains leaves up to l_b. We then have, inductively, $f(s_{a-1}, i_{a:b}) = s_b$ and $f(s_b, i_{b+1:c}) = s_c$, which gives $f(f(s_{a-1}, i_{a:b}), i_{b+1:c}) = s_c$. By the combining rule we have $f(s_{a-1}, g(i_{a:b}, i_{b+1:c})) = s_c$, and by associativity we have have $f(s_{a-1}, i_{a:c}) = s_c$. This claim proves property (2).
Claim 2: If t_b is the leaf with the least index in the subtree of a node n, then the state returned to the node n by its parent (or by the memory in case of the root) is s_{b-1}.
Proof by induction on the depth of the node: The hypothesis is clearly true of the root. If a node n is the left child of its parent, then it receives the same state that its parent received and the transaction with the least index is the same for both, which makes the claim valid. If n is the right child of its parent, consider its left sibling n'. Let t_a be the transaction of the lowest index in the subtree rooted at

```
int FA_f(int s, int a) {return s + a;}

int CAS_v_f(int v, int s, int vn) {
  return (v == s) ?  vn :  s; }

int CAS_v_h(int v, int s, int vn) {
  return (v == s) ?  1 :  0; }
```

Figure 3: Combining functions for FA and CAS_v.

n'. Then the parent of n received state s_{a-1}. Therefore n receives state $f(s_{a-1}, i_{a:b-1}) = s_{b-1}$ by claim 1.

We can now prove property (1) by induction on the index of the transaction using the second claim: property (1) is clearly true of t_1, as it receives $h(s_0, i_1)$ by claim 2. Suppose the property is true of t_j for some j, $1 \leq j < k$. Let n be the lowest common ancestor of t_j and t_{j+1}, and let n_L, n_R be the left and right children of node n. t_j is the transaction with the highest index in the subtree rooted at node n_L. Therefore, by claim 2, n_R receives state s_j. Because all ancestors of t_{j+1} below n_R are the left children of their parents, it receives the same state as n_R and therefore has output $h(s_j, i_{j+1})$. □

In this paper we define the functions e, f, g and h associated with ϕ in C code using an interface of the form:

```
iitype phi_e(itype)
stype phi_f(stype, iitype)
iitype phi_g(iitype, iitype)
otype phi_h(stype, itype)
```

where `iitype` is the extended input type (I'). When we omit either the `phi_e` function or the `phi_h` function it means it is the identity function (for h it is the identity projection on the state). When we omit the `phi_g` function it is the same as the `phi_f` function (and $S = I'$). In cases in which the combining function is parameterized based on some value, we add the type `ptype` as the first argument to these functions, if it is needed. Figure 3 defines the required functions for the FA and CAS_v as defined in Figure 1. For both FA and CAS_v `iitype` matches the other types (`int`). As an example, here we show that these are indeed the correct combining functions for CAS_v. We note that when the CAS_v operations combine in a tree the leftmost request with a value (`vn`) not equal to v is the one that swaps with memory (if memory contains v).

THEOREM 2. *The CAS_v operations are combinable with the functions specified in Figure 3.*

PROOF. We need to show the four properties: projection, combining, associativity and bounded size. We have

$$f(a,b) = g(a,b) = \begin{cases} b & a = v \\ a & \text{otherwise} . \end{cases}$$

We then have:

combining. $f(f(s_0, v_1), v_2) = f(s_0, g(v_1, v_2))$

associativity. $g(g(v_1, v_2), v_3) = g(v_1, g(v_2, v_3))$

which both follow from the associativity of f (and g). Projection is trivial since e is the identity and bounded size is true since all values are of type `int`. □

We note that although the CAS_v is combinable, the standard multi-valued CAS is not. The problem is that all values that need to be compared must be passed on, because the combining function g cannot know which one might match the memory value. This violates the bounded size condition. Many algorithms from the literature use a multi-value CAS and therefore cannot take advantage of combining. Thus for combining, one seeks opportunities to replace CAS with a CAS_v or other combinable MBT (e.g., see Section 4.1). Note that the combinable CAS_v is sufficient for the wait-free consensus protocol and hence is universal for wait-free simulation [9].

Cost Model. For correctness we will assume that any instruction can take an arbitrary finite amount of time, and we require that all code is correct under this assumption. For an MBT x the time taken is $t'_x - t_x$. We refer to the maximum time taken by any instruction as τ and analyze algorithms and protocols in terms of τ. We assume at any time there could be a mixture of types of transactions being applied to any memory location, some that combine and some that do not. Given high contention and no combining, τ could be large because the current accesses to a location would be serialized. One way to address this is to account for contention in the cost model apart from τ [3, 6]. Here, instead, we will define a notion of bounded contention, and will only consider protocols that have bounded contention.

We say that two transactions *contend* if their blocks overlap but they are not compatible. Consider a transaction history H and let $H_{l,t} \subset H$ be all transactions x that involve location l and are active at time t (*i.e.*, $t_x \leq t \leq t'_x$). The *contention* for a location l and a time t is the number of equivalence classes in $H_{l,t}$ based on compatibility. A protocol or algorithm has *bounded contention* if, for all possible histories with any number of processors, the maximum contention at any location and time is bounded by some fixed constant.

4. EXAMPLES

We now consider several examples of protocols and algorithms that can be implemented using MBTs. All the MBTs we use are combinable. For each we prove various bounds that in some way improve over the known bounds for the problem considered.

4.1 Priority Write

Our first example is a priority write that succeeds in writing a value dependent on a priority. A *prioritized value* of type t is a pair consisting of a priority and a value of type t. A priority write with prioritized value (p, v) will overwrite a prioritized value (p', v') in memory if $p > p'$. It returns 1 if the write succeeds and 0 otherwise. Figure 4 defines the priority write using a MBT on a structure `pval` consisting of an integer priority `pr` and pointer value `v`. It also defines the f and h functions for the combinable form. It is easy to see that these functions satisfy the required projection, combining, associativity, and bounded size properties. Note that in these and other code segments throughout this paper, struct types are assigned and returned as a unit, e.g., `*p = a` indicates that both fields of a are copied into the struct referenced by `p`.

A priority write can be particularly useful when many processors are writing to the same location in parallel and the

```
struct pval {int pr; void* v};

atomic int pWrite(pval *p, pval a) {
  if (a.pr > p->pr) {
    *p = a;
    return 1; }
  else return 0; }
```

(a)

```
pval pWrite_f(pval a, pval b) {
  return (b.pr > a.pr) ?  b :  a; }

int pWrite_h(pval a, pval b) {
  return (b.pr > a.pr) ?  1 :  0; }
```

(b)

Figure 4: Priority write (`pWrite`) for pointer types (`void *`): (a) the MBT, and (b) the f and h functions for the combinable version.

user wants the value with the highest priority to win. There are many applications in shared-memory algorithms [12]. For example in a minimum-spanning-tree algorithm each vertex might want to determine its minimum adjoining edge (the so-called Borùvka step). This can be implemented by having each edge write to both endpoints using priority `pr` = 0 − edge weight. Because the degree of a vertex could be high, this can involve many writes to the same location. In Section 4.4 we use a priority write to support concurrent writes in our transactional memory (TM) system. The idea is that each TM transaction has a timestamp and serializability is guaranteed by using priority writes to have the TM transaction with the latest timestamp succeed in writing.

Here we consider how a priority write can be used to replace a "versioned" CAS [11]—a two word CAS that overwrites the current value and increments the version if neither has changed since the last read. Such a versioned CAS is often used to avoid the so-called ABA problem [11] in which a value changes and changes back. Because the version continuously increases, the CAS will recognize that the value has been written even when it returns to an old value. A versioned CAS can be implemented with a double-word CAS with the version in one word and the value in the other. The problem with the CAS used in this context is that we cannot use the combinable CAS_v. Moreover, there can be many processors each with a different version writing to the same location so the contention is not bounded. By replacing the CAS with a priority write, we can bound the contention.

We present an example of replacing a versioned CAS with a priority write in the context of Treiber's well-known lock-free memory-allocation scheme [29]. Treiber's algorithm allocates and frees nodes by keeping them in a free list—an allocation removes an element from the front of the list and a free returns the element to the front. Allocating and freeing both involve reading from the head of the list and writing back a modified version. To ensure the operations are atomic, Treiber's implementation uses a versioned CAS so that the new value is written into the head only if it has not changed since it was read.

In our variant we replace the CAS with a priority write. The code for `alloc` and `free` is shown in Figure 5. The head of the list is stored in the variable `head`, which stores a priority (version) and a pointer to the head. The priority keeps a version number that is incremented by one every time the head is written. As with a CAS the priority write will succeed only if no other user has written the head since the head was read. Unlike the CAS, however, the priority write can be combined since all we care about is the maximum version being written. Using the CAS forces the system to send all the versions to the memory even though only the latest version could possibly succeed.

```
struct node {void *val; node *next;};

pval head;

node *alloc() {     /* pop */
  do {
    pval x = head;
    node *n = x.v;
    x.pr = x.pr + 1;
    x.v = n->next;
  } while (!pWrite(&head,x));
  return n; }

void free(node *n) {     /* push */
  do {
    pval x = head;
    n->next = x.v;
    x.pr = x.pr + 1;
    x.v = n;
  } while (!pWrite(&head,x)); }
```

Figure 5: Variant of Treiber's linearizable stacks for lock-free memory allocation, using priority write instead of CAS.

In addition to using a priority write, the implementation in Figure 5 uses a concurrent read of the head. Concurrent read is a trivial combinable MBT.

We note that, as with Treiber's algorithm, the solution suffers the problem that if the version numbers overflow then the algorithm fails. For 64-bit integers, over 10^{19} updates would be required to overflow the counter. This can be relatively easily fixed at the cost of giving up the lock-free property in the rare cases that the version overflows—access can be blocked until all current operations complete and then the version can be reset to 0.

THEOREM 3. *The implementation of* `alloc` *and* `free` *of Figure 5 has the following properties assuming unbounded priorities:*

1. *it is linearizable,*
2. *it is lock-free,*
3. *it has bounded contention, and*
4. *if some user is executing an* `alloc` *or* `free` *then some user will finish within $O(\tau)$ time.*

PROOF. Note that the `alloc` and `free` operations will complete only after their `pWrite` succeeds. As noted above, a `pWrite` succeeds only if no other operation has updated the head in the time interval between the read and the `pWrite`. Moreover, the head is updated only by a successful `pWrite`. Consider a history $\mathcal{H}$ at some time t. The order of the successful `pWrite` transactions specifies a proper linearization order among the associated `alloc` and `free` operations in $\mathcal{H}$. The linearization order includes all completed operations in $\mathcal{H}$ and is consistent with both $<_{\mathcal{H}}$ and the sequential semantics of the operations. Because the combining functions for `pWrite` (and for concurrent read) satisfy the projection, combining, associativity, and bounded size properties, property (1) follows from Theorem 1.

Properties (2) and (4) follow from the observation that the `pWrite` fails on two consecutive loop iterations only if another `alloc` or `free` operation succeeds in the meantime. Finally, property (3) follows from the observation that the implementation has only a constant number of distinct combinable operations on the shared variables. □

4.2 Semaphores

The second example we consider is an implementation of Dijkstra's (counting) semaphores. The example demonstrates how a modified version of fetch-and-decrement that does not decrement below zero (NN-FD) can be used to implement a simple non-blocking version of the P operation. This seems hard without this variant of fetch-and-add. With semaphores the $P(S)$ and $V(S)$ operations are used to share a limited resource: the $P(S)$ operation is used to enter the critical section for semaphore S and $V(S)$ to exit it. We assume that a user exits the critical section before trying to enter it again. If a semaphore S is initialized to k then up to k users can be in the critical region simultaneously, but no more. The $P(S)$ operation blocks when the semaphore is full. Gottlieb et al. [8] describe the following elegant implementation of the $P(S)$ and $V(S)$ operations using fetch-and-add operations:

```
void P(int S) {
  while (1) {
    if (S > 0)
      if (FA(S,-1) > 0) return;
      else FA(S,1); } }

void V(int S) { FA(S,1); }
```

The implementation allows concurrent access to S and has bounded contention. It is sometimes, however, useful to also support a non-blocking $tryP(S)$ operation that enters and returns 1 if the semaphore is not full, but otherwise returns 0 immediately. Gottlieb et al. did not consider such an operation, but one might try the following implementation.

```
int tryP(int S) {
  if (S > 0)
    if (FA(S,-1) > 0) return 1;
    else FA(S,1);
  return 0; }
```

Unfortunately this implementation is not linearizable. In particular the following sequences by two users with $S = 1$ initially can lead to the last $tryP()$ by each user failing (returning 0).

```
   U1          U2
tryP(S)     tryP(S)
            V(S)
            tryP(S)
```

Such behavior does not correspond to any sequential interleaving of the operations. The problem arises when the `FA(S,-1)` of the first $tryP$ on `U2` occurs between the `S > 0` test and the `FA(S,-1)` of the $tryP$ on `U1`. This can cause `U1` to temporarily decrease S to -1 even though it will later readjust it back up (via its `FA(S,1)`) and return failure. This temporary decrease will cause the second $tryP$ on `U2` to fail under the following scenario: If both the V and the $tryP$ on `U2` complete prior to the readjustment by `U1`, then the V sets S to 0 and the `S > 0` test fails for the $tryP$ on `U2`. We see no easy way to fix this with a fetch-and-add, or even also using a CAS if we want to maintain concurrent access.

```
atomic int NNFD(int *p) {
  int v = *p;
  *p = (v > 0) ?  v-1 :  0;
  return v; }
```

(a)

```
int NNFD_e() {return 1;}

int NNFD_f(int S, int a) {
  return max(S-a,0);}

int NNFD_g(int a, int b) {
  return a + b;}
```

(b)

Figure 6: `NNFD` decrements its argument, but not past zero: (a) the memory-block transaction, and (b) the definition of e, f and g for the combinable version (h is the identity projection on state).

This problem can easily be fixed using an NN-FD. An MBT for such an operation is defined in Figure 6, along with the associated functions for the combinable version. Note that the e function makes explicit the implicit decrement amount (i.e., 1). One can readily show that e, f, g, and h satisfy the projection, combining, associativity, and bounded size properties.

Based on this NN-FD the code for `tryP` can be written as:

```
int tryP(int S) {
  if (NNFD(S) > 0) return 1;
  else return 0; }
```

The $P(S)$ operation can also be simplified to

```
int P(int S) {while (!NNFD(S));}
```

whereas the $V(S)$ operation remains the same. As an aside, note that the full generality of NN-FD is not required because we only test whether the value returned is zero or greater than zero.

In the following theorem we say a user is in the *critical region* if the user has executed a $P(S)$ (or a successful $tryP(S)$) that has responded but the user has not since invoked a $V(S)$.

THEOREM 4. *The `NNFD` implementation of $P(S)$, $V(S)$ and $tryP(S)$ with S initialized to k has the following properties:*

1. *it is linearizable,*
2. *it has bounded contention, and*
3. *$V(S)$ and $tryP(S)$ both respond in $O(\tau)$ time, and if some user is waiting on $P(S)$ and fewer than k users are in the critical region then some $P(S)$ or $tryP(S)$ will respond within $O(\tau)$ time.*

PROOF. *Property (1):* Consider a history $\mathcal{H}$ of $P(S)$, $tryP(S)$ and $V(S)$ operations. Let $\mathcal{H}'$ be the subset of $\mathcal{H}$ comprised of all $P(S)$ operations with a successful `NNFD(S)`, all $tryP(S)$ operations with a completed `NNFD(S)` (successful or otherwise), and all $V(S)$ operations with a completed `FA(S,1)`. Note that $\mathcal{H}'$ contains all completed operations and all operations that have updated S. $\mathcal{H}'$ is linearizable in the following order: order the $P(S)$, $tryP(S)$ and $V(S)$ operations based on the order in which their successful `NNFD(S)`, their completed `NNFD(S)`, and their completed `FA(S,1)`, respectively, were applied at the memory. Among

those operations in $\mathcal{H}'$ that were applied at the memory simultaneously due to compatible combining, order them as dictated by Theorem 1.

Property (2): The implementation has bounded contention because it consists of only two distinct non-combinable operations (FA and NN-FD) on the semaphore.

Property (3): The implementation maintains the invariant that the value `v` of the semaphore S indicates the difference between k and the number of users currently in the critical section. If the number of such users is less than k, then S has a non-zero value, which implies that some `NNFD` (invoked by the waiting user or otherwise) will succeed immediately. Thus some $P(S)$ or $tryP(S)$ will respond in $O(\tau)$ time. □

4.3 FIFO Queue

Our next example is a linearizable bounded FIFO queue. The example demonstrates a somewhat more sophisticated example of atomically incrementing and checking for overflow/underflow to achieve a queue implementation with good properties.

There has been a long history of concurrent algorithms for the FIFO queue, some have turned out to be incorrect, and most have some limitation. Perhaps the best known is Michael and Scott's algorithm [22]. It is lock-free but assumes an unbounded queue and hence that infinite memory has been allocated ahead of time. It also sequentializes access to the queue. Gottlieb et al. [8] (GLR) describe an array-based algorithm that supports bounded queues and concurrent access using a FA. However, it is not lock-free and more importantly it is not linearizable [2]. Blelloch et al. [2] (BCG) describe a linearizable version that supports bounded queues with $O(\tau)$ access time. It is based on *room synchronizations* that separate enqueues and dequeues in time. Thus, a user may need to wait until the next round for its enqueue or dequeue and BCG is not lock-free.

The algorithm we describe here has the same properties as the BCG queue but in the common case does not require any waiting. It is based on a modified version of the GLR queue. The problem with the GLR algorithm with respect to linearizability is similar to the problem with semaphores described in Section 4.2. In particular it uses a FA to keep a current upper and lower bounds on the size of the queue. When the queue overflows on an enqueue or is empty on a dequeue, the FA temporarily sets the bounds to an inconsistent value, restoring consistency only in a subsequent step. This temporary state breaks linearizability. We see no way of avoiding this problem using a standard FA operation while allowing enqueues and dequeues to happen concurrently. The solution we present here is to atomically update the top and bottom queue pointers while also checking for under or overflow.

We use MBTs on a state consisting of three fields: the bottom pointer (where dequeues are taken from), the top pointer (where enqueues are added), and the queue size. On this state we use two MBTs, an `enqLoc` that returns the enqueue position or -1 if the queue overflows, and a `deqLoc` that returns the dequeue position or -1 if the queue is empty. If the queue overflows on an enqueue or is empty on a dequeue then the state is left untouched. Figure 7 defines the state (`qloc`) and the MBTs. The locations continuously increase but are mapped into a bounded queue of the specified size using modular arithmetic. Note that `q->bot` $\leq$ `q->top` $\leq$ `q->bot+q->size`.

```
struct qloc {int bot; int top; int size};

atomic int enqLoc(qloc *q) {
  int overflow = q->bot + q->size;
  int top = q->top;
  if (top >= overflow) return -1;
  else {
    q->top = top + 1;
    return top; }}

atomic int deqLoc(qloc *q) {
  int bot = q->bot;
  if (bot >= q->top) return -1;
  else {
    q->bot = bot + 1;
    return bot; } }
```

(a)

```
int enqLoc_e() { return 1; }

qloc enqLoc_f(qloc s, int cnt) {
  qloc q = s;
  q.top = min(q.bot + q.size,
              q.top + cnt);
  return q; }

int enqLoc_g(int a, int b) {return a + b;}

int enqLoc_h(qloc s) {
  if (s.bot + s.size > s.top) return s.top;
  else return -1; }
```

(b)

```
int deqLoc_e() { return 1; }

qloc deqLoc_f(qloc s, int cnt) {
  qloc q = s;
  q.bot = min(q.top,
              q.bot + cnt);
  return q; }

int deqLoc_g(int a, int b) {return a + b;}

int deqLoc_h(qloc s) {
  if (s.top > s.bot) return s.bot;
  else return -1; }
```

(c)

Figure 7: Atomic operations `enqLoc` and `deqLoc` for getting the location for an enqueue or dequeue for a FIFO Queue, while checking for underflow or overflow: (a) defines the MBTs, (b) and (c) define e, f, g, and h for the combinable versions.

Using `enqLoc` and `deqLoc` a linearizable FIFO queue can be implemented as shown in Figure 8. The queue wraps around when it reaches the top. Although the algorithm is not lock-free it has the property that in the common case the enqueue and dequeue do not block. A block can only be caused either when the dequeue or enqueue wraps around (infrequent for large enough queue) or when the operation has to wait for the location to fill or empty (unlikely if the processors are working at the same speed especially if the queue is not near full or empty). The block for wraparound is caused by the first while loop and the block for fill or empty by the second.

THEOREM 5. *`enqLoc` and `deqLoc` are combinable.*

PROOF. We detail the proof for `enqLoc`. The proof for `deqLoc` is similar.

projection: The e function for `enqLoc` is the constant function 1. Both `enqLoc_f` on input 1 and `enqLoc` do the same

```
struct queue {qloc loc;
              void *A[];
              int enqDone;
              int deqDone; }

int enqueue(queue *q, void *v) {
  int s = (q->loc).size;
  int i = enqLoc(q->loc);
  if (i < 0) return 0;
  else {
    while (i/s > q->enqDone/s);
    while(q->A[i%s] != NULL);
    q->A[i%s] = v;
    FA(q->enqDone,1);
    return 1; } }

void *dequeue(queue *q) {
  int s = (q->loc).size;
  int i = deqLoc(q->loc);
  if (i < 0) return NULL;
  else {
    while (i/s > q->deqDone/s);
    while(q->A[i%s] == NULL);
    void *result = q->A[i%s];
    q->A[i%s] = NULL;
    FA(q->deqDone,1);
    return result; } }
```

Figure 8: Implementation of a linearizable bounded-size FIFO Queue with $O(\tau)$ time access.

operation on the state: add 1 to `top` if the queue is not overflowing, i.e., if `bot+size` is smaller than `top`.

combining: $f(s, g(i_1, i_2))$ increases the value of `top` by the minimum of $(i_1 + i_2)$ and `size-(top-bot)`. If i_1 is smaller than `size-(top-bot)` in s, then $f(s, i_1)$ increases `top` in s by i_1 and $f(f(s, i_1), i_2)$ increases `top` in $f(s, i_1)$ by the minimum of i_2 and `size-(top-bot)`-i_1. If i_1 is larger than `size-(top-bot)` in s, then $f(s, i_1)$ increases `top` in s by `size-(top-bot)` and $f(f(s, i_1), i_2)$ keeps `top` in $f(s, i_1)$ unchanged. Therefore, in both cases, $f(f(s, i_1), i_2)$ increases `top` in state s by the minimum of (i_1+i_2) and `size-(top-bot)`, which is what $f(s, g(i_1, i_2))$ does.

associativity: both $g(g(i_1, i_2), i_3)$ and $g(i_1, g(i_2, i_3))$ yield $i_1 + i_2 + i_3$.

bounded size: $|I'|$ is the length of an integer which is smaller than $|S|$. □

THEOREM 6. *The implementation of a FIFO queue as defined in Figure 8 with size s has the following properties:*

1. *it is linearizable,*
2. *it has bounded contention, and*
3. *all enqueue and dequeue operations with r concurrent requests finish in $O(\lceil r/s \rceil \tau)$ time.*

PROOF. Let $<_l$ be the linearized order of the `enqLoc` and `deqLoc` calls made by the enqueue and dequeue operation. Let E_s (D_s) be the set of successful enqueues (dequeues), i.e., those that do not overflow (underflow). We claim that $<_l$ is a valid linearization for the enqueues and dequeues and show this by arguing that under this ordering the following properties of a queue hold:

Q1. An enqueue operation x succeeds *iff* $|\{e <_l x | e \in E_s\}| - |\{d <_l x | d \in D_s\}| < s$.

Q2. A dequeue operation x succeeds *iff* $|\{e <_l x | e \in E_s\}| > |\{d <_l x | d \in D_s\}|$.

Q3. The i^{th} successful dequeue receives the value from the i^{th} successful enqueue.

In the implementation, only a successful enqueue (dequeue) operation can increase the variable `qloc->top` (`qloc->bot`) by 1, and this variable is not changed anywhere else. Therefore, the difference `(qloc->top - qloc->bot)` is exactly the difference between the number of prior successful enqueue and dequeue operations. Thus, the check `(top>=overflow)` in `enqLoc` ensures that property Q1 holds (similar for property Q2).

Let `i` be the index returned by `enqLoc` or `deqLoc` in a successful enqueue or dequeue. For a successful enqueue x we have `i` $= |\{e <_l x : e \in E_s\}|$ and similarly for dequeues (same reason as above). All enqueues write into location `i%s` and all dequeues read from location `i%s`. We say an enqueue or dequeue *is in round* p if $\lfloor$`i/s`$\rfloor = p$.

Property Q3 holds if a dequeue operation in round p always reads the value written to A[`i%s`] by an enqueue in round p. To ensure this, the implementation prevents enqueues (dequeues) in round `i+1` from writing (reading) until all enqueues (dequeues) in round `i` have written (read). It does this by keeping track of the number of completed enqueue (dequeue) operations in the variable `enqDone` (`deqDone`) and only incrementing it after the write (read). The check (`i/s > q->enqDone/s`) then prevents an enqueue from writing until the previous round is all written (similarly for dequeues). The check (`q->A[i%s] != NULL`) in the enqueue ensures that the dequeue from round $p-1$ at `A[i%s]` has removed its value before writing a new value. Similarly the check (`q->A[i%s] == NULL`) in the dequeue ensures that the enqueue from round p has written its value before reading the value. Thus, property Q3 holds.

Therefore, $<_l$ is a valid linearization, and by Theorems 5 and 1, the implementation is linearizable.

Bounded Contention. Bounded contention follows from the fact that the implementation has only a constant number of distinct combinable operations on the shared variables.

Run time. An enqueue operation either receives a positive value `i` or the value -1 from `enqLoc` in $O(\tau)$ time. If it receives -1, the enqueue function returns and thus completes in $O(\tau)$ time. If not, the difference between `i` and the current value of `enqDone` is at most r, and therefore, it has to wait for at most $\lceil r/s \rceil$ rounds before it can enter the value into the array and return.

Claim: After all the enqueue operations from round p are done, all the waiting enqueue operations from round $p+1$ complete in $O(\tau)$ time.

To show this, note that the atomic operations ensure that an enqueue operation is assigned round $p+1$ and array location j only if some dequeue operation has already been assigned round p and array location j (otherwise the enqueue operation would have failed). This dequeue cannot be blocked, because (1) all dequeues in round $p-1$ (if $p > 0$) must be done otherwise the enqueues in p could not be done, and (2) the corresponding enqueue in round p is done. Therefore the dequeue will complete in time $O(\tau)$ and the waiting enqueue in round $p+1$ will complete in $O(\tau)$.

From the fact that any operation has to wait for at most $\lceil r/s \rceil$ rounds and that each round takes $O(\tau)$ time, it follows that any enqueue finishes in $O(\lceil r/s \rceil \tau)$ time. This can correspondingly be shown for dequeues. □

```
typedef int tstamp;

struct pval {tstamp pr; void* v};

struct loc {
  tstamp readT;
  tstamp writeT;
  int writeCnt;
  void *v};

atomic void *readLoc(loc *s, tstamp t) {
  if (s->writeT <= t && s->writeCnt == 0) {
    s->readT = max(t,s->readT);
    return s->v:
  } else return NULL; }

atomic int writeReserve(loc *s, tstamp t) {
  s->writeCnt++;
  if (s->readT <= t) {
    return 1;
  } else return 0; }

atomic void writeCommit(loc *s, pval a) {
  s->writeCnt--;
  if (a.pr > s->writeT) {
    s->writeT = a.pr;
    s->v = a.v; } }

atomic void writeRelease(loc *s) {
  s->writeCnt--; }
```

Figure 9: Read and Write operations for Transactional Memory.

4.4 Transactional Memory

Our next example is a scheme and set of operations for supporting software transactional memory (STM) [26]. The scheme is based on concurrency control using single-version timestamp ordering [25, 1]. Every transaction is given a unique *timestamp* when it starts (easily implemented with a FA), and each transactional memory location stores a pair of timestamps, corresponding to the latest user transactions that read and that wrote the location. Our implementation uses MBTs to avoid the need for locks to update timestamps—they are updated atomically on a user read and write. The scheme is meant as an example of the use of MBTs and not as a complete design of a viable STM system.

In addition to allowing concurrent reads and concurrent writes to the same location, the scheme we describe requires only a single access to memory for each read and guarantees all reads seen by a transaction form a valid snapshot of the state. Many optimistic concurrency schemes do not guarantee a valid snapshot unless they re-check all previous reads on every read [18].

The MBTs we use are shown in Figure 9. Mutable shared values used in transactions are stored in a `loc` structure. Each write is partitioned into a reserve and a commit phase, similar to a two-phase lock. The protocol for a transaction works as follows:

1. Acquire a timestamp.
2. Execute user transaction code using `readLoc` for reads and locally buffering all writes. If any `readLoc` fails, abort the transaction.
3. Use `writeReserve` to reserve all buffered writes. If any `writeReserve` fails, release all reserved locations using `writeRelease` and abort the transaction.
4. Use `writeCommit` to commit all buffered writes.

```
loc readLoc_f(loc s, tstamp t) {
  if (s.writeT <= t && s.writeCnt == 0)
    s.readT = max(t,s.readT);
  return s; }

tstamp readLoc_g(tstamp a, tstamp b) {
  return max(a,b);}

void *readLoc_h(loc s, tstamp t) {
  if (s.writeT <= t && s.writeCnt == 0)
    return s.v;
  else return NULL; }
```

Figure 10: Combining functions for `readLoc`.

```
int writeReserve_e (tstamp t) {
  return 1; }

loc writeReserve_f (loc s, int n) {
  s.writeCnt += n; return s; }

int writeReserve_g (int a, int b) {
  return a+b; }

int writeReserve_h (loc s, tstamp t) {
  if (s.readT <= t)
    return 1;
  else return 0; }
```

Figure 11: Combining functions for `writeReserve`.

Aborting a transaction will cause it to retry and acquire a new timestamp. We say that a user read or write is at time t if it is part of a user transaction with timestamp t. Within each `loc` the `readT` (`writeT`) field indicates the latest time the location was read (written, respectively). The `writeCnt` indicates the number of transactions that have attempted to reserve a location but have not yet released or committed a write to it. A read at time t will fail *iff* either the latest write to that location happened after t, or the location has been attempted for write reservation and has not yet been released (`writeCnt` > 0). A write at time t will fail *iff* the latest read to the location happened after t. The `writeCommit` uses a priority write to allow concurrent writes based on the Thomas write rule [28, 1]—a write will be ignored if it happens before the current version. In the `readLoc` code we assume a failed memory read is indicated by returning a NULL pointer. Alternatively this can be implemented using an exception mechanism.

One will note that four words is a lot to store just one word of user data. We imagine, however, `loc`s would be used to store pointers to user objects. Also the two timestamps could be short words (e.g. 4 bytes) and the `writeCnt` is needed solely to support concurrent writes. Therefore if concurrent writes are not necessary, a `loc` could be stored in 16 bytes (8 bytes for data, and 8 bytes for the two timestamps). As with using priority writes with versioning, the timestamps can overflow. This can be handled by blocking transactions that would overflow the timestamp, waiting for all existing transactions to complete, and then traversing memory to reset all timestamps to zero.

We say two transactions conflict if they overlap in time and reference the same variable. As with standard usage, these can be read-read, read-write, or write-write conflicts.

THEOREM 7. *The described implementation of user transactions has the following properties:*

1. *only a read-write conflict can cause an abort,*

```
struct mpval {tstamp pr; void* v; int cnt;};

mpval writeCommit_e (pval a) {
  mpval ret;
  ret.pr = a.pr; ret.v = a.v; ret.cnt = 1;
  return ret; }

loc writeCommit_f (loc s, mpval a) {
  s.writeCnt -= a.cnt;
  if (a.pr > s.writeT) {
    s.writeT = a.pr; s.v = a.v; }
  return s; }

mpval writeCommit_g (mpval a, mpval b) {
  mpval c;
  if (a.pr >= b.pr) {
    c.pr = a.pr; c.v = a.v; c.cnt = a.cnt+b.cnt;
  } else {
    c.pr = b.pr; c.v = b.v; c.cnt = a.cnt+b.cnt; }
  return c; }
```

Figure 12: Combining functions for `writeCommit`.

```
int writeRelease_e () { return 1; }

loc writeRelease_f (loc s, int a) {
  s.writeCnt -= a; return s; }

int writeRelease_g (int a, int b) {
  return a+b; }
```

Figure 13: Combining functions for `writeRelease`.

2. *successful transactions are serializable based on the timestamps they are allocated,*
3. *all reads seen by a transaction are consistent with some point in the serialized state, and*
4. *the protocol has bounded contention.*

PROOF. First note that the e, f, g and h functions described above for `readLoc`, `writeReserve`, `writeCommit` and `writeRelease` satisfy the projection, combining, associativity and bounded size properties making these functions combinable. We include the proof that `readLoc` satisfies the four properties:

projection: This property follows from the fact that function e for `readLoc` is the identity function and that `readLoc` and `readLoc_f` are identical except for the part of `readLoc` that returns a value.

combining: The only part of the state that `readLoc` can alter is `readT`. $f(s, g(i_1, i_2))$ replaces `s.readT` with the greater of `s.readT` and the two inputs (timestamps) if the condition `c` $\equiv$ `(s.writeT <= t && s.writeCnt==0)` is true for state s on either of the inputs. f when applied to s with input i_1 attempts to replace `s.readT` only if condition `c` holds. Because the truth value of condition `c` for state s with input i_2 does not change upon the application of f to s, `s.readT` is replaced in $f(f(s, i_1), i_2)$ only if condition `c` is satisfied by state s either on input i_1 or i_2. Thus, in both $f(f(s, i_1), i_2))$ and $f(s, g(i_1, i_2))$, `readT` is replaced under the same conditions. In case `s.readT` is replaced in $f(f(s, i_1), i_2)$, only the maximum among the two inputs would finally appear for the value of `s.readT`, which makes the replacement identical with that of $f(s, g(i_1, i_2))$.

associativity: Both $g(g(i_1, i_2), i_3)$ and $g(i_1, g(i_2, i_3))$ return the maximum integer among i_1, i_2 and i_3 in this case.

bounded size: $|I'|$ is smaller than $|S|$.

We now prove properties (1)–(4) in turn.

Property (1): An abort is caused only when either (i) a `writeReserve` operation encounters a read timestamp set ahead of its timestamp, or (ii) a `readLoc` encounters an attempt to write in the form of `writeReserve` whose corresponding `writeCommit` or `writeRelease` has not yet been called. These cases occur only when there is a read-write conflict.

Property (2): First note that a block of memory is written to only by a successful user transaction. Because only successful user transactions can change the contents of user memory, it is safe to ignore any attempts made by unsuccessful transactions at writing to a memory block. The fact that successful transactions are serializable in the order of their timestamps follows from this claim:

Claim: A successful `readLoc` request invoked by a user transaction T with timestamp t to a memory block B returns either (i) the value written to B by the successful transaction T' with the highest timestamp $t' \leq t$ that writes to B, if such a transaction exists, or (ii) the initial state of memory block B, otherwise.

Case (i): When transaction T reads from B, it reads the value written to B (using a `s.v = a.v` step) by the latest combined group of `writeCommit` operations that wrote a value into B before T attempted a read. We want to show that the `writeCommit` request of transaction T' was grouped into the last such group and that T' has the largest timestamp among all transactions in that group. Suppose that this were not the case. Then either (a) a different group of `writeCommit` operations with highest timestamp t_a (corresponding to transaction T_A) has written to B after T' and before T reads B, (b) T' performed `writeCommit` after T's read request, or (c) the `writeCommit` request of T' was combined along with the last group of `writeCommit`s that wrote to B before T's `readLoc` invocation reached memory, but another transaction T_C with timestamp $t_c > t$ had its `writeCommit` request combined along with that of T'. If (a) had occurred, then in order for T_a to have written to memory, $t_a > t'$ (because of the check `a.pr > s.writeT`). Also, since T has performed a successful read operation after T_a has committed a write, $t_a \leq t$ (because of the check `s.writeT <= t`). This violates the assumption that T' is the transaction with the highest timestamp $t' \leq t$ that has written to B, thus ruling out possibility (a). Now consider (b): if the `writeReserve` request of T' reached memory before `readLoc` request of T, then T would have aborted (because of the check `s.writeCnt==0`). On the other hand, if `writeReserve` request of T' reached memory after `readLoc` request of T, then the reserve `writeRequest` would have failed (because of the check `s.readT<=t`). This rules out possibility (b). In order for T's `readLoc` to succeed in possibility (c), $t_c \leq t$ (because of the check `s.writeT<=t`). This violates the assumption that T' is the transaction with the highest timestamp $t' \leq t$ that has written to B.

Case (ii): If there is no transaction with timestamp $\leq t$ that writes to B, the only situation in which `readLoc` of T does not return the initial state of memory location B is when a transaction with timestamp $t_2 > t$ has written to B before `readLoc` of B reached memory. But in such a situation, transaction T would abort as its `readLoc` fails.

Property (3): All reads are consistent with the serializable state determined by the order of the transaction timestamps. To see this, note that the earlier claim is true of

both successful and unsuccessful transactions. All successful read requests of a transaction with timestamp t read off values that were supposed to be in the requested memory blocks at time t in the serialized state based on the order of timestamps.

Property (4): Bounded contention follows from the fact that the implementation has only a constant number of distinct combinable operations on the shared variables. □

5. RELATED WORK

Our formalism for combinable memory-block transactions is derived from a long history of previous work. The idea of partitioning the transition function into the two parts f and g to enable combining goes back at least to Kogge and Stone's work on solving recurrence equations in parallel [15]. A similar and elegant framework was also used by Kruskal, Rudolph and Snir for defining combinable read-modify-write operations [16]. The idea of combining in hardware also has a long history. In addition to the already cited work on the Ultracomputer other machines with some support for atomic updates in the memory system include the IBM RP3 design [23] and the Connection Machine II [27] (combining in the network), and the SGI Origin 2000 [19] and Cedar [17] (atomic operations at the memory controller).

6. DISCUSSION

Although the emphasis of this paper is not on how to implement MBTs, here we briefly describe three implementation issues: specification of transition functions, combining, and interaction with cache coherence.

Transition functions. We have not discussed how the functions ϕ, f and g might be sent along with an MBT. One possibility is to define a compact coding scheme that enables the actual code to be sent along with the request. We expect that all functions given in this paper could be coded in one 8-byte word. Another solution is to download the functions into the memory system and then specify a pointer to the function as part of the memory request. This could reduce the number of bits needed to send with the request but introduces various other issues—how are functions protected among domains; how are they distributed to every memory module; or what happens when the program space runs out? For these reasons the first approach seems more attractive especially given that 8-bytes compared to the cache-line size does not seem onerous.

Another issue with transition functions is what if the function takes too long or does not terminate? This might be dealt with either by not allowing code with loops or by having a time-limit. With a time-limit the memory would need to be able return an exception.

Combining. Combining as suggested in the Ultracomputer [8, 16] was based on the idea that memory requests for the same location would combine when they happen to run into each other in a router queue within a switch in the network. Only a single combined request is then forwarded and the switch maintains the required state until the reply from memory returns. This requires a comparator within a switch that can detect when messages within a queue are destined for the same destination. The advantage of such a system is that it is exactly when requests start to get congested and the queues back up that requests to the same location meet in the switches and combine. This approach was justified theoretically [14, 24] requiring only that either the queues in each switch are maintained using a priority order on the requests [14], or that requests are sent in a priority order [24].

Modern architectures, however, are almost all designed around a cache hierarchy and it is not clear how combining might work in such an organization. We note without systematic justification that shared caches at each level of the cache hierarchy might be an ideal place to combine requests. Imagine, for example, a cache on a chip that is shared among p processors and that there is bottleneck for memory requests to get off the chip. This would likely require requests to back up within the cache, ultimately stalling the processors. We note, however, that the cache maps requests for the same memory location to the same cache line giving a natural way to detect requests destined for the same location and combine them. Furthermore the cache provides a natural place to store the state required for a combined request while waiting for the response. If all processors are updating the same variable pointing to the head of a queue, for example, these requests could be combined in the cache and only one request forwarded. This could be repeated at the next level of the cache hierarchy.

Cache coherence. A concern with MBTs is how they might interact with memory coherence schemes. One possibility is to not cache lines that are involved in MBTs. This can be done either (1) by separating the memory address space and not caching some regions, as was done by the SGI Origin 2000 [19] to support certain atomic operations at the memory controller, or (2) by acquiring a coherent copy of the cache line at the home memory controller before applying the transaction there, as suggested in the AMO paper [4]. Either approach integrates well with combining—in the latter case the combined request would acquire the coherent copy. Both approaches work well when the location has poor locality or heavy write sharing but not very well under light-load where locality is important.

In the transactional memory scheme we describe, for example, most reads and writes might never interact and having all requests go to their home node could require excessive latency. To avoid this one could allow processors (caches) to take exclusive writable copies and then execute the MBT at the cache, but it is not clear how this would be integrated with combining. It seems, therefore, that the best solution is to allow the MBT to be applied either at the home or at the local cache depending on the load. For example, whenever a combined request arrives, the home could acquire ownership and execute the MBT. A timer might also be used to detect the last request from a different location and only allow taking an exclusive copy if sufficient time has passed.

Acknowledgments

Blelloch and Vardhan were supported in part by NSF Grant CCR-0122581, a gift from Intel and by Microsoft as part of the Carnegie Mellon Center for Computational Thinking. We thank the reviewers for their helpful comments on the paper.

7. REFERENCES

[1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[2] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable room synchronizations. *Theory Comput. Syst.*, 36(5):397–430, 2003.

[3] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6):779–805, 1997.

[4] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *Proc. 21st ACM Int'l Conf. on Supercomputing*, pages 232–241, 2007.

[5] F. E. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional primitives. *Distrib. Comput.*, 18(4):267–277, 2006.

[6] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theor. Comput. Sci.*, 196(1-2):3–29, 1998.

[7] A. Gottlieb, R. Grishman, C. P. Kruskal, C. P. Mcauliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing an MIMD parallel computer. *IEEE Trans. Comput.*, C-32(2):75–89, 1983.

[8] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, 1983.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[11] IBM T.J. Watson Res. Ctr. System/370 principles of operations. Technical report, IBM, 1983.

[12] J. Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[13] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[14] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *J. ACM*, 35(4):876–892, 1988.

[15] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. on Computers*, C–22(8):786–793, 1973.

[16] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.

[17] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, 231:967–974, 1986.

[18] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2007.

[19] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proc. 24th ACM Int'l Symp. on Computer Architecture*, pages 241–251, 1997.

[20] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):41–61, 1993.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[22] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[23] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research parallel processor prototype (RP3): Introduction and architecture. In *Proc. IEEE Int'l Conf. on Parallel Processing*, pages 764–771, 1985.

[24] A. G. Ranade. How to emulate shared memory. *J. of Comput. Syst. Sciences*, 42:307–326, 1991.

[25] R. M. Shapiro and R. E. Millstein. Rehability and fault recovery in distributed processing. In *Oceans '77 Conf Record, vol II*, 1977.

[26] N. Shavit and D. Touitou. Software transactional memory. *Distrib. Comput.*, 10(2):99–116, 1997.

[27] Thinking Machines Corporation. Model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, 1987.

[28] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proc. IEEE COMP-CON Conf.*, 1978.

[29] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.